High-Order Stencil Computations on Multicore Clusters

Liu Peng, Richard Seymour, Ken-ichi Nomura, Rajiv K. Kalia, Aiichiro Nakano, Priya Vashishta
Collaboratory for Advanced Computing and Simulations, Department of Computer Science, Department of Physics & Astronomy, Department of Chemical Engineering & Material Science, University of Southern California, Los Angeles, CA 90089-0242, USA
(liupeng, rseymour, knomura, rkalia, anakano, priyav)@usc.edu

Alexander Loddoch, Michael Netzband, William R. Volz, Chap C. Wong
Technical Computing, Chevron ETC, Houston, TX 77002, USA
(loddoch, mknetzband , Bill.Volz, ChapWong)@chevron.com

Abstract

Stencil computation (SC) is of critical importance for broad scientific and engineering applications. However, it is a challenge to optimize complex, high-order SC on emerging clusters of multicore processors. We have developed a hierarchical SC parallelization framework that combines: (1) spatial decomposition based on message passing; (2) multithreading using critical section-free, dual representation; and (3) single-instruction multiple-data (SIMD) parallelism based on various code transformations. Our SIMD transformations include translocated statement fusion, vector composition via shuffle, and vectorized data layout reordering (e.g. matrix transpose), which are combined with traditional optimization techniques such as loop unrolling. We have thereby implemented two SCs of different characteristics—diagonally dominant, lattice Boltzmann method (LBM) for fluid flow simulation and highly off-diagonal (6-th order) finite-difference time-domain (FDTD) code for seismic wave propagation—on a Cell Broadband Engine (Cell BE) based system (a cluster of PlayStation3 consoles), a dual Intel quadcore platform, and IBM BlueGene/L and P. We have achieved high inter-node and intra-node (multithreading and SIMD) scalability for the diagonally dominant LBM: Weak-scaling parallel efficiency 0.978 on 131,072 BlueGene/P processors; strong-scaling multithreading efficiency 0.882 on 6 cores of Cell BE; and strong-scaling SIMD efficiency 0.780 using 4-element vector registers of Cell BE. Implementation of the high-order SC, on the contrary, is less efficient due to long-stride memory access and the limited size of the vector register file, which points out the need for further optimizations.

1. Introduction

Stencil computation (SC) is a common kernel of a wide range of scientific and engineering applications such as Jacobi and multigrid solvers [1]. Examples include explicit time-integration methods for numerical solution of partial differential equations used in climate, weather and ocean modeling [2], computational electromagnetics [3] and quantum dynamics [4] codes using the finite-difference time-domain (FDTD) method, multimedia/image-processing applications that perform smoothing and other neighbor pixel-based computations [5], and certain cellular automata and seismic simulations [6]. Because of its significance, SC is included in a number of benchmark suites, such as SPEC [7], HPFBECH [8], and NAS Parallel Benchmarks [9].

SC involves a variable $v(\mathbf{R})$ defined for a set of discrete grid points $\{\mathbf{R}\}$ in a regular lattice, and it sweeps over $\{\mathbf{R}\}$ to update $v(\mathbf{R})$ as a function of the values, $v(\mathbf{R}')$, of neighbor grid points $\mathbf{R}' \in neighbor(\mathbf{R})$:

```
for $\forall \mathbf{R} \in \{\mathbf{R}\}$
$v(\mathbf{R}) \leftarrow f(\{v(\mathbf{R}') | \mathbf{R}' \in neighbor(\mathbf{R})\})$
```

where f is a mapping function. A typical SC thus consists of doubly nested loops: The outer loop sweeps over all grid points $\mathbf{R}$ in the lattice to update $v(\mathbf{R})$; and the inner loop over $\mathbf{R}' \in neighbor(\mathbf{R})$ computes $f(\{v(\mathbf{R}')\})$. SC may be classified according to the spatial pattern of $neighbor(\mathbf{R})$ as follows. First, the order of a stencil is defined as the distance between the central grid point $\mathbf{R}$ and the farthest grid points $\mathbf{R}'$ in $neighbor(\mathbf{R})$ along a certain axis. (In a finite-difference application, this order is identical to that of Taylor expansion.) Second, we define the size of a stencil as the cardinality $|neighbor(\mathbf{R})|$, i.e. the number of grid points **R**′ in *neighbor*(**R**) including **R** itself. In diagonally dominant SC, the computation depends heavily on the local value *v*(**R**) as well as some *v*(**R**′) values from the nearest-neighbor grid points. Figure 1(a) shows a diagonally dominant, first-order, 19-point SC in the 3-dimensional lattice Boltzmann method (LBM) flow simulation [10, 11]. SC in other applications is often much higher order involving distant neighbor grid points. For example, Fig. 1(b) shows a 6-th order, 25-point SC in a 2-dimenional lattice. Such stencil is widely used in high-order finite-difference calculations [12, 13].

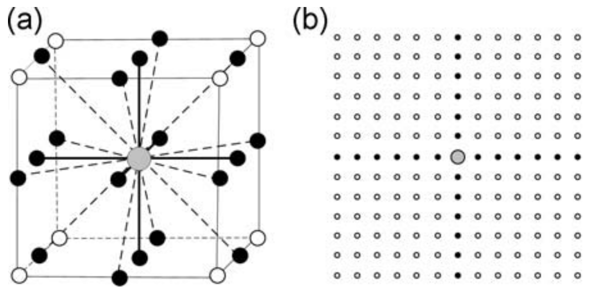

Figure 1. (a) Diagonally dominant, first-order, 19-point stencil in a 3D lattice for LBM flow simulation, and (b) 6-th order, 25-point stencil in a 2D lattice. For a grid point R (gray circle), the neighbor grid points R′ ∊ *neighbor*(R) are shown as solid circles, while the rest of the grid points as open circles.

Emerging parallel platforms comprised of multicore compute nodes potentially provide enormous computing power for SC. An example is a cluster of general microprocessors consisting of multiple cores such as Intel quadcore processor. Another example is Cell Broadband Engine (Cell BE) in PlayStation3 and QS22 blade [14]. It is a heterogeneous multicore chip consisting of a traditional microprocessor called power processing element (PPE) that controls eight single-instruction multiple-data (SIMD) co-processing units called synergistic processing elements (SPEs) [15].

There have been extensive efforts to optimize SC on multicore platforms. Williams et al. [16] have optimized the LBM application on single Cell BE blade. Traditional SC optimization techniques include tiling [17] and iteration skewing (if the iteration structure allows it) [1, 18-20]. Recently, Datta et al. have performed comprehensive optimization of SC with both cache-aware and cache-oblivious approaches [21]. These studies are mainly focused on low-order SC. Since both SC and multicore architectures have wide varieties as mentioned above, it is desirable to develop a unified parallelization framework and perform systematic performance optimization for various SCs (e.g., diagonally dominant vs. high-order) on multicore architectures.

This paper presents our hierarchical SC parallelization framework that combines: (1) spatial decomposition based on message passing; (2) multithreading using critical section-free, dual representation; and (3) single-instruction multiple-data (SIMD) parallelism based on various code transformations. We employ SIMD transformations such as translocated statement fusion, vector composition via shuffle, and vectorized data layout reordering (e.g. matrix transpose), which are combined with traditional optimization techniques such as loop unrolling. We thereby parallelize two SCs of different characteristics—diagonally dominant (first order, 19-point) LBM for fluid flow simulation, and high-order (6th-order, 37-point) FDTD for seismic wave propagation. The two SC applications have been implemented on a Cell BE based platform (PlayStation3 cluster), dual Intel quadcore, and IBM BlueGene/L and P. Section 2 presents the hierarchical SC parallelization framework applied to LBM, followed by test results of inter- and intra-node scalability on a PlayStation3 cluster and IBM BlueGene/L and P, as well the effect of SIMD, multithreading and their combination on a PlayStation3 cluster. In section 3, we discuss the optimization of the seismic wave propagation code on a dual Intel quadcore platform. Section 4 contains conclusions drawn from our experiments.

2. Hierarchical Parallelization of Lattice Boltzmann Method

In the lattice Boltzmann method (LBM) for fluid flow simulation, variable *v*(**R**) is a density function (DF) from which fluid density and velocity at **R** are computed. The LBM simulation consists of timing-stepping iterations, where each step consists of two phases: (1) collision function that involves a large number of floating-point operations that are strictly local to each **R**; and (2) streaming function that contains no floating-point operation but solely memory copies between nearest-neighbor grid points.

Our hierarchical parallel Lattice Boltzmann Method (pLBM) algorithm parallelizes LBM based on spatial decomposition implemented with message passing, multithreading using critical section-free, dual representation, and SIMD parallelism, which maximally exposes concurrency and data locality. The following subsections describe these parallelization layers.

2.1. Spatial Decomposition Based on Message Passing

In the uppermost level of the hierarchical spatial decomposition in our pLBM algorithm, the total simulation space $\Omega = \{\mathbf{R}\}$ is decomposed into several spatial sub-domains Ω_i, where $\Omega = \cup_i \Omega_i$ and each domain is mapped onto a processor. Figure 2 shows a two-dimensional example, where a single domain consists of $N_x \times N_y$ grid points (N_x and N_y are the numbers of lattice sites in the x and y directions, respectively). Each domain is augmented with a surrounding buffer layer used for inter-domain DF migrations. We impose a boundary condition on DFs propagating toward the closed nodes (i.e. spatial regions with obstacles that exclude fluid flow), by reflecting back DFs propagation into the closed nodes toward the opposite direction. (The actual code implements a 3D LBM with the 19-point stencil in Fig. 1.)

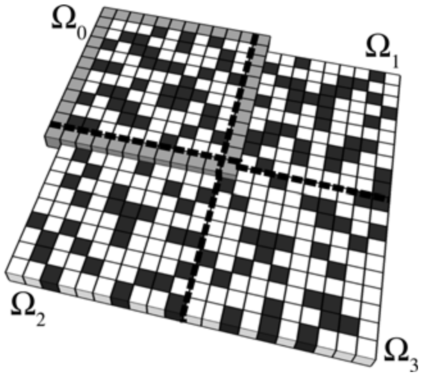

Figure 2. Schematic of spatial decomposition in 2 dimensions with 4 domains (Ω_0-Ω_3). White squares are open lattice sites that have the DFs of flow particles. Black squares represent obstacles, where flow is excluded. Gray squares are buffer-layer sites for Ω_0.

We have implemented the spatial decomposition using the message passing interface (MPI) standard and have tested the inter-node scalability on the IBM BlueGene/L computer at the Lawrence Livermore National Laboratory and the IBM BlueGene/P computer at the Argonne National Laboratory. Figure 3(a) shows the running and communication times of the pLBM code on up to 212,992 BlueGene/L and 131,072 BlueGene/P processors. Here, we scale the number of lattice sites linearly with the number of processors: 128^3P lattice sites on P processors. The computing time increases only slightly when P increases from 1 to 131,072 on BlueGene/P. The weak-scaling parallel efficiency is the running time on 1 processor divided by that on P processors, and it is 0.978 on 131,072 BlueGene/P processors. Nearly ideal efficiency is also evident on 212,992 IBM BlueGene/L processors in Fig. 3(a). Spatial decomposition is thus highly effective in terms of inter-node scalability for SC up to 10^5 compute nodes.

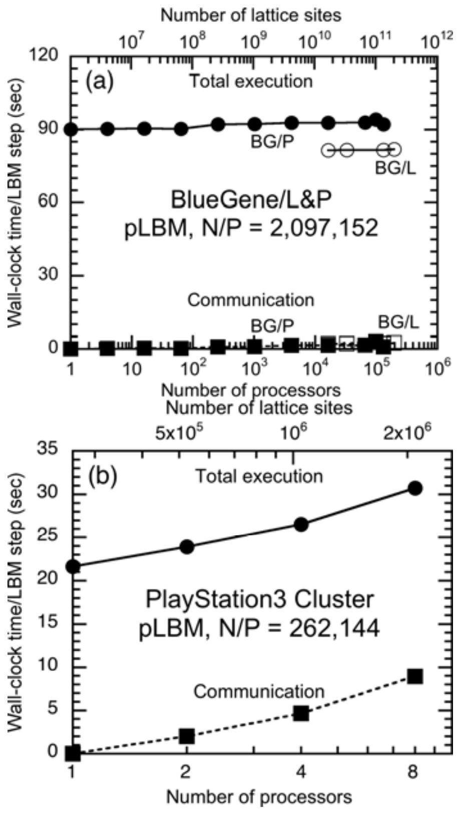

Figure 3. Running (circles) and communication (squares) times of the pLBM flow simulation as a function of the number of processors on: (a) BlueGene/L (open symbols) and BlueGene/P (solid symbols), where each processor is assigned 128^3*P* lattice sites; and (b) PlayStation3 cluster with 64^3*P* lattice sites per PlayStation3 console.

Figure 3(b) shows the inter-console parallel efficiency of pLBM on a cluster of 8 PlayStation3 consoles. Here, we scale the number of lattice sites linearly with the number of processors: 64^3P lattice sites on P processors. Despite the small bandwidth and large latency of the low-cost Gigabyte Ethernet switch, the parallel efficiency of the PlayStation3 cluster is respectable (0.704 on 8 consoles).

2.2. Multithreading Based on Critical Section-Free, Dual Representation

On a cluster of PlayStation3 (PS3) consoles, pLBM is implemented based on hierarchical spatial decomposition: (1) inter-console parallelization with the upper-level spatial decomposition into domains based on message passing; and (2) intra-console parallelization with the lower-level spatial decomposition into interleaved rows of the lattice sites within each domain through multithreading. Inside each PS3 console, the main program running on the power processing element (PPE) spawns multiple threads to run on synergistic processing elements (SPEs). Data transfer between the main memory of PPE and the local storage of SPEs is handled by direct memory access (DMA).

For the collision function, 6 SPE programs are performed simultaneously with 6 threads created by PPE (due to the PS3 hardware restriction, only 6 SPEs out of 8 are available for user programming). For optimal load balancing, we adopt an interleaving schema shown in Fig. 4(a), where the area enclosed by the dotted lines shows the computational task assigned to the first thread with thread ID 0, and chunk ID j is assigned to SPE with thread ID j mod N_{thread}, $j \in [0, N_{\mathbf{thread}} - 1]$ (the number of threads N_{thread} is 6). Since multiple threads may update a common lattice site, we adopt a double-layered DF consisting of two floating-point arrays *DF0* and *DF1* to avoid any critical section, as shown in Fig. 4(b): The collision function transfers DFs from array *DF0* to local store on SPE, updates the DFs, and subsequently copies it back to the array *DF1*.

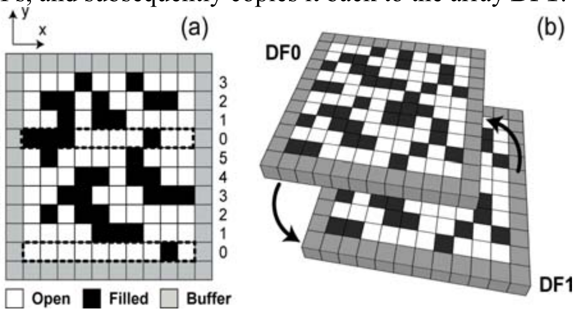

Figure 4. (a) Schematic of a 2-dimensional system setup for each domain in spatial decomposition. White squares are open lattice sites that have the DF's, black squares represent obstacles, where flow does not exist, and gray squares are buffer-layer sites. The simulation system is divided into N_y computational chunks, each of which consists of N_x lattice sites, and the chunks are interleavingly assigned to SPEs. The numerals show thread-ID responsible for each chunk. (b) Schematic of a double-layered DF calculation comprising two floating-point arrays *DF0* and *DF1*. The collision function reads DFs from *DF0* to do updates and then stores the result in *DF1*. Subsequently, the streaming function propagates DFs from *DF1* to *DF0*.

We have tested the intra-node (or multithreading) scalability of pLBM on each PlayStation3 console. Figure 5(a) shows the running time of pLBM involving 64^3 lattice sites as a function of the number of SPEs (i.e. threads). The corresponding strong scaling (or fixed problem-size) speedup (circles) in Fig. 5(b) is close to the ideal speedup (solid line). The algorithm thus scales nearly linearly with the number of SPEs. On 6 SPEs, the speedup is 5.29, and the parallel efficiency (defined as the speedup divided by the number of threads) is 0.882.

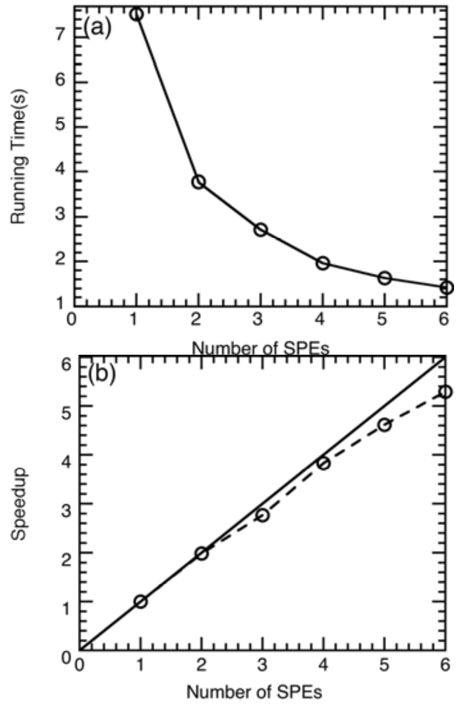

Figure 5. Running time (a) and strong scaling speedup (b) of pLBM as a function of the number of SPEs on a PlayStation3 console.

2.3. Single-Instruction Multiple-Data Parallelism

Most modern computing platforms have incorporated single-instruction multiple-data (SIMD) extensions into their processors [22] to exploit the natural parallelism of applications if the data can be SIMDized (i.e., if a single instruction can simultaneously operate on a vector of consecutive data). On Cell BE, each vector contains four floating-point numbers that are operated concurrently, and thus the ideal speedup is 4. We maximally expose SIMD parallelism of pLBM using various code transformations. The following subsections describe our SIMD transformations: Translocated instruction fusion, vector composition via shuffle, and vectorized data layout reordering (e.g. matrix transpose).

2.3.1 Translocated Statement Fusion for SIMD

The key to SIMD parallelization is to identify SIMDizable statements, i.e., a set of statements that are independent and thus are executable concurrently, when the associate variables are packed into vectors. Statements that satisfy this criterion can be transformed into a single SIMD statement regardless of the positions of the statements (e.g., in different loops) in the program. The translocation of statements and their fusion into a SIMD operation enhance the concurrency of the program at the instruction level. In the following subsection, we illustrate the use of such translocated statement fusion for SIMD parallelization in pLBM.

Here, the original code is doubly nested for loops, where the inner loop traverses the 18 nearest neighbor grid points in a cube to perform certain computation:

```
for(ic=0;ic<3;ic++){
  l=0;
  rho=0.0;
  for(u[ic]=0.0;l<18;l++){
  fi[l] =
buffers[buf idx].f[18*incz+l];
  u[ic] += fi[l]*v[l][ic];
  rho += fi[l];
  }
}
```

Our SIMD solution (Fig. 6) first exchanges the order of the two for loops, and then unrolls the inner loop over ic to 0, 1, and 2. Subsequently, we exploit four floats, $v[l][0]$, $v[l][1]$, $v[l][2]$ and rho, which are SIMDizable (i.e., they can be multiplied and added simultaneously) and can be looped for 18 times each. We pack them into a vector

$\{v[l][0], v[l][1], v[l][2], \text{rho}\}$,

generate another vector

$\{fi[l], fi[l], fi[l], fi[l]\}$,

and perform vector multiple and vector add instructions on them. This is repeated by a for loop to get the final vector, which is $\{u[0], u[1], u[2], \text{rho}\}$.

The resulting SIMDization can be expressed as

$$\{u[0], u[1], u[2], \text{rho}\} = \sum_{l=0}^{17} \{fi[l], fi[l], fi[l], fi[l]\} * \{v[l][0], v[l][1], v[l][2], \text{rho}\}.$$

The speedup of this SIMDization can be analyzed as follows. The original program performs 3×18×(2+2+1) = 270 operations by counting both multiplication "*" and addition "+" as one, while the SIMDized program requires only 18×(2+2) = 72 operations. Thus the speedup due to SIMD is estimated as 3.75.

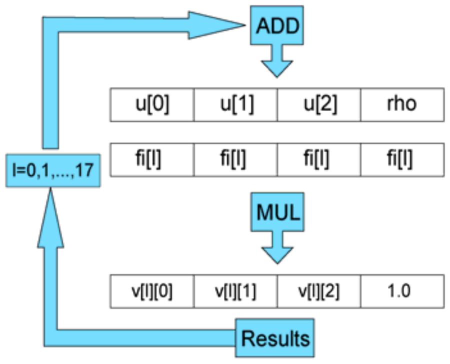

Figure 6. Example of SIMDization by translocated statement fusion in pLBM.

2.3.2 Vector Composition for SIMD

The flexibility of composing new vectors is another useful feature of SIMD supported by shuffle and extract instructions. The shuffle(**vec1**, **vec2**, **pattern**) function is a byte-oriented operation that selects bytes according to the pattern (a control vector) from source vectors **vec1** and **vec2** and places them in a target vector [23]. The control vector entries are indices of bytes in the 32-byte concatenation of **vec1** and **vec2**: To select byte n from **vec1**, we define the corresponding byte of the pattern as "0X0n", while from vec2 as "0X1n". Figure 7 illustrates the use of shuffle to generate a new vector $\mathbf{uvec_0} = \{u[0], u[0], u[0], u[0]\}$ from vector $\mathbf{uvec} = \{u[0], u[1], u[2], u[3]\}$. By assigning **uvec** to **vec1**, **vec2** and defining pattern0 as in Fig 7, we can select desired bytes of the **uvec** to generate **uvec0**.

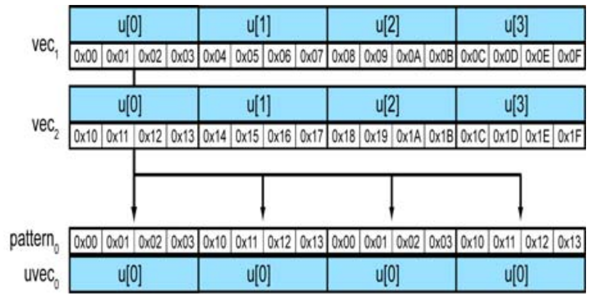
vec1
u[0]
u[1]
u[2]
u[3]
0x00 0x01 0x02 0x03 0x04 0x05 0x06 0x07 0x08 0x09 0x0A 0x0B 0x0C 0x0D 0x0E 0x0F
vec2
u[0]
u[1]
u[2]
u[3]
0x10 0x11 0x12 0x13 0x14 0x15 0x16 0x17 0x18 0x19 0x1A 0x1B 0x1C 0x1D 0x1E 0x1F
pattern0
0x00 0x01 0x02 0x03 0x10 0x11 0x12 0x13 0x00 0x01 0x02 0x03 0x10 0x11 0x12 0x13
uvec0
u[0]
u[0]
u[0]
u[0]

Figure 7. Example of shuffle to compose a new vector in pLBM.

Here, the original code is a doubly nested for loops to calculate all $u[i]*u[j]$, where both i and j indices run from 0 to 2:

```
for(i=0;i<3;i++)
  for(j=0;j<3;j++)
    pieq[i][j]=u[i]*u[j];
```

Our SIMD solution first packs all u[i] s into vector uvec as **uvec** = {u[0], u[1], u[2], u[3]}, where u[3] is set as NULL to fill the vector. We then define three patterns and perform three shuffle operations with different patterns to obtain three new vectors:

$\mathbf{uvec_i} = \{u[i], u[i], u[i], u[i]\}$ (i = 0, 1, 2).

Subsequently, we multiply **uveci** (i = 0, 1, 2) by uvec, and put the results into **pieqveci** (i = 0, 1, 2), respectively. The pseudo-code of the above SIMDization is given below:

```
uvec = {u[0],u[1],u[2],u[3]}
for(i=0;i<3;i++){
uveci=shuffle(uvec,uvec,patterni);
pieqveci = mul(uvec,uveci);
}
```

The speedup of this SIMDization can be analyzed as follows. While the original program performs 3×3×1 = 9 operations, the SIMD program requires only 3. Thus the speedup is 3. Memory access is analyzed as well. While the original program accesses memory 3×3×3 = 27 times, the SIMD program only requires 1+3×1 = 4 memory accesses. Thus the memory access time is reduced by a factor of 7 by the SIMDization. The use of array of structure (AOS), i.e., the packing of 3 floats into a vector leaving the fourth element NULL, incurs some overhead, which is largely offset by the SIMD speedup.

It is highly beneficial to generate new data from existing vector registers instead of fetching them from cache or memory each time, and this greatly improves the data reuse ratio and reduces the memory access time. Data generation flexibility of SIMD via vector composition thus not only reduces computation but also reduces memory access.

2.3.3 Data Layout Reordering for SIMD

Data layout reordering is another SIMDizing technique facilitated by shuffle. By defining appropriate patterns, for instance, it is possible to transpose the rows and columns of an array efficiently. Figure 8 shows a transpose function for a 4×4 array, TRANSPOSE4*4(v0,v1,v2,v3), where each **vi** (i = 0,1,2,3) is a vector. The transpose procedure in Fig. 8 consists of two steps, both of which are shuffle operations. First, the 4×4 array is arranged as four vectors, where each row itself is a vector. We then define patterns for shuffle operation, and execute the set of four shuffles in Step 1 (Fig 8), followed by another set of four shuffles in Step 2 (Fig 8) to obtain the transposed array. Since all the vectors are in the vector register, only four memory accesses are involved, making this SIMDized transpose highly efficient.

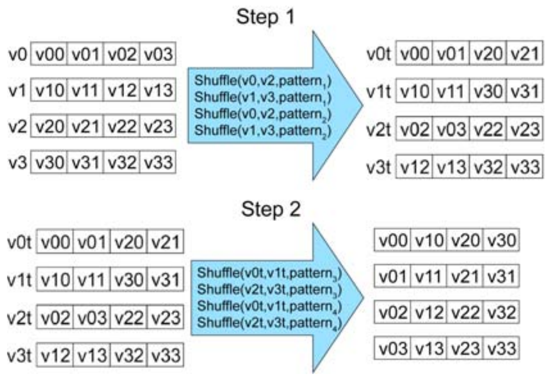

Figure 8. Transpose of a 4×4 array by shuffle operations for SIMDization.

We use this SIMDized transpose to parallelize pLBM. Here, the original code is a doubly nested for loops to do something like matrix multiplication:

```
for(j=0;j<3;j++){
  pi[i][j] = 0.0;
  for(l=0;l<18;l++)
    pi[i][j] += fi[l]*v[l][j];
}
```

The inner loop accesses an 18×3 array, v, column by column. The resulting stride memory accessing degrades the performance considerably and necessitates data layout reordering. Our SIMD solution resolves this problem in a two-step procedure: (1) transpose—v is transposed using divide-and-conquer, first dividing it into 5 small arrays and then using TRANSPOSE4*4() to transpose each sub-array; and (2) calculation—the transpose reduces the problem to the one described in section 2.3.1, which is SIMDized using the same transposed statement fusion schema. The pseudo-code of the above SIMDization of transpose is given below:

```
TRANSPOSE18*3(v[][]){
  for(l=0,l<18,l=+4)
TRANSPOSE4*4(v[l],v[l+1],v[l+2],NULL);
}
```

The speedup of the SIMDized transpose of an 18×3 array can be analyzed as follows. The original program requires 3×18×(1+2) = 162 operations, while the SIMD program operates 5×4×2 = 40. The speedup is thus 2.7. This is remarkable considering the overhead of the SIMDization. First, we transpose an array of 4×4 each time, while the real program only has 4×3 data. Also, the TRANSPOSE18*3 function in fact transposes 20×3 data instead of 18×3.

Using the techniques described above, we have SIMDized pLBM, where the SIMDization was only implemented to the collision function, which accounts for the most computing time of pLBM. We first test the SIMD effect with various problem sizes both for the collision part (which is fully SIMDized) and the whole pLBM application on a PS3, where the number of threads is set to one. Figure 9(a) shows the SIMD speedup of the collision function (i.e. the ratio of the running time without SIMD and that with SIMD). It is an ascending function of the problem size and reaches 3.72 for 2.61×105 grid points, which is close to the ideal speedup of 4 (SIMD efficiency is 0.930). Figure 9(b) shows the SIMD effect for the whole pLBM application. The SIMD speedup of pLBM is again an ascending function of the problem size and reaches 3.12 for 2.614×105 grid points (SIMD efficiency 0.780). The SIMD optimization is thus highly effective for pLBM.

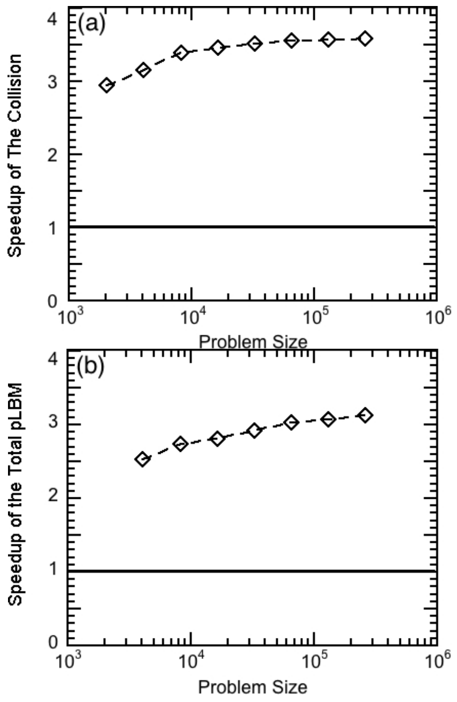

Figure 9. Effect of SIMD with problem size on a PS3. The speedup of collision function due to SIMD (a) and the speedup of pLBM due to SIMD (b) are plotted as a function of the problem size.

Next, we test the SIMD speedup for different numbers of threads for the collision part and the whole pLBM application on a PS3 for 2.614×105 grid points. Figure 10(a) shows that the speedup for the collision (i.e. pure SIMD) part is nearly constant around 3.5 with varying number of threads. The SIMD optimization thus has very good scalability with different numbers of threads. However, for the whole pLBM application, the speedup is a decreasing function of the number of threads and reduces to 2.1 for 6 threads (i.e. the maximum number of threads on a PS3). This is because both communication and streaming-computation times do not decrease for larger numbers of threads. Consequently, the overall speedup for the whole pLBM is not as good as for the pure SIMD part.

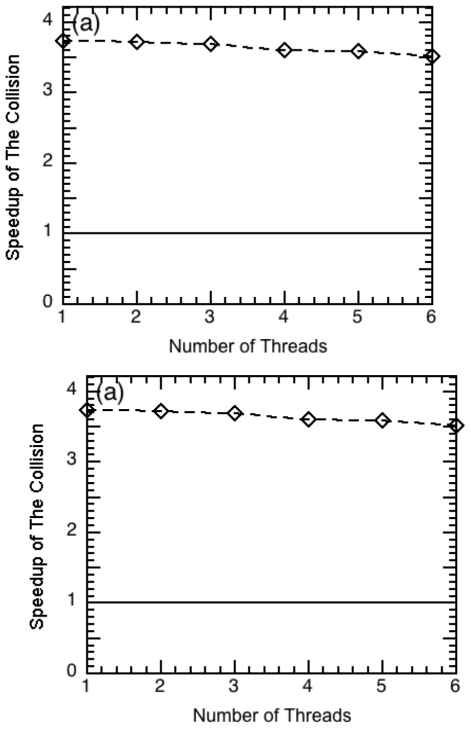

Figure 10. SIMD speedup for different numbers of threads on a PS3 for (a) collision function and (b) the entire pLBM.

3. High-Order Seismic Stencil Computation on Dual Intel Quadcore

We have applied the same parallelization framework as described in section 2 to a finite-difference time-domain (FDTD) code for seismic wave propagation, which performs a large stencil computation with typical problem sizes of $1,024^3$ grid points. This section describes some specific performance features of the FDTD code. The 3D stencil computation here is highly off-diagonal (6-th order) and involves 37 points, i.e., each grid point interacts with 12 other grid points in each of the x, y and z Cartesian directions. The original FDTD code is written with standard POSIX threads to run on x86 multicore systems, and we choose a dual Intel quadcore platform as a testbed. Spatial decomposition divides the system into 2D layers, and each thread works on its layer in the x direction, taking 12 inputs from the main array outputting a data point into a temporary array. Then the temporary array is run through again with a 12-point stencil and sent into the final array. For the y and z directions, the initial 3D array is transposed before being operated on, and the final array is then summed out in strides in order to return to the 3D arrangement.

3.1. Optimization Schema for Seismic Code

As mentioned above, the 37-point SC in the seismic code accesses 12 neighbor points in each Cartesian direction, in order to update the value of one grid point. To find the bottleneck of the code, we have run the code with the Intel VTune profiler on a dual Intel quadcore platform, and the result shows that the most time-consuming parts are nine computations and two stride memory accessing. Each of the nine computations (three for each of the x, y, z directions) contains nested for loops over central and neighbor grid points. The two stride memory accessing transpose the 3D array data to make the memory access in the y and z directions contiguous, respectively, for the computations.

We have applied the SIMDization approaches described in section 2.3 to the parallel seismic code. Here, we illustrate the procedure using a representative code segment shown below, which calculates one value from 12 nearest neighbors:

```
for(i=istart;i<iend;i++){
   bD[i+5]=-(b[i+5]+b[i+6])*
   (a6*(u[i]-u[i+11])
   +a5*(u[i+1]-u[i+10])
   +a4*(u[i+2]-u[i+9])
   +a3*(u[i+3]-u[i+8])
   +a2*(u[i+4]-u[i+7])
   +a1*(u[i+5]-u[i+6]));
}
```

Our SIMD solution first unrolls the loop four times (we choose four for alignment), and then makes use of the translocated statement fusion and vector composition for SIMD as described in section 2.3. Prior to the for loop, we pack floats u[i] to u[i+11] to three vectors, uvec_0, uvec_1, and uvec_2:

$$\mathbf{uvec_k} = \{u[i+4k], u[i+4k+1], u[i+4k+2], u[i+4k+3]\}\ (k = 0, 1, 2).$$

We also pack the constants a_k (k = 1, 2, … ,6) to three vectors:

$$\mathbf{avec_0} = \{a_6, a_5, a_4, a_3\},$$
$$\mathbf{avec_1} = \{a_2, a_1, -a_1, a_2\},$$
$$\mathbf{avec_2} = \{-a_3, -a_4, -a_5, -a_6\}.$$

First, we initialize $\mathbf{utmp}_k$ as $\mathbf{uvec}_k$ (k= 0, 1, 2). For the first loop, we multiply each $\mathbf{utmp}_k$ with $\mathbf{avec}_k$ (k=0, 1, 2) and store the product to vector sumvec. After calling horizontal addition hadd (as defined in Fig. 11(b)) twice, we extract the first float value of sumvec with extract. Subsequently, the float value is multiplied by b to get a bD. Next, we load $\mathbf{uvec}_3 = \{u[i+12], u[i+13], u[i+14], u[i+15]\}$. For the second loop, we first use shuffle operations to get new $\mathbf{utmp}_0$, $\mathbf{utmp}_1$, $\mathbf{utmp}_2$, and do the same computation to get another bD. Similar operations are repeated in the third and the fourth loops except for different patterns for shuffle. After the fourth loop, the four bDs are written back as a vector and then update $\mathbf{uvec}_0$, $\mathbf{uvec}_1$, $\mathbf{uvec}_2$ with $\mathbf{uvec}_1$, $\mathbf{uvec}_2$, $\mathbf{uvec}_3$ for use in the next round. The pseudo-code below and Fig. 11 illustrate the SIMDization:

```
subroutines:
 compute(bD[i+5]){
 sumvec=avec0*utmp0+avec1*utmp1+avec2*
 utmp2;
    sumvec=hadd(sumvec);
    sumvec=hadd(sumvec);
    bD[i+5]=-(b[i+5]+b[i+6])
    *extract(sumvec);
 }
 shuffle(patterni){
utmp0=shuffle(uvec0,uvec1,patterni);
utmp1=shuffle(uvec1,uvec2,patterni);
utmp2=shuffle(uvec2,uvec3,patterni);
 }

 main function:
 pack uveci and aveci (i=0,1,2)
 define patterns:
```

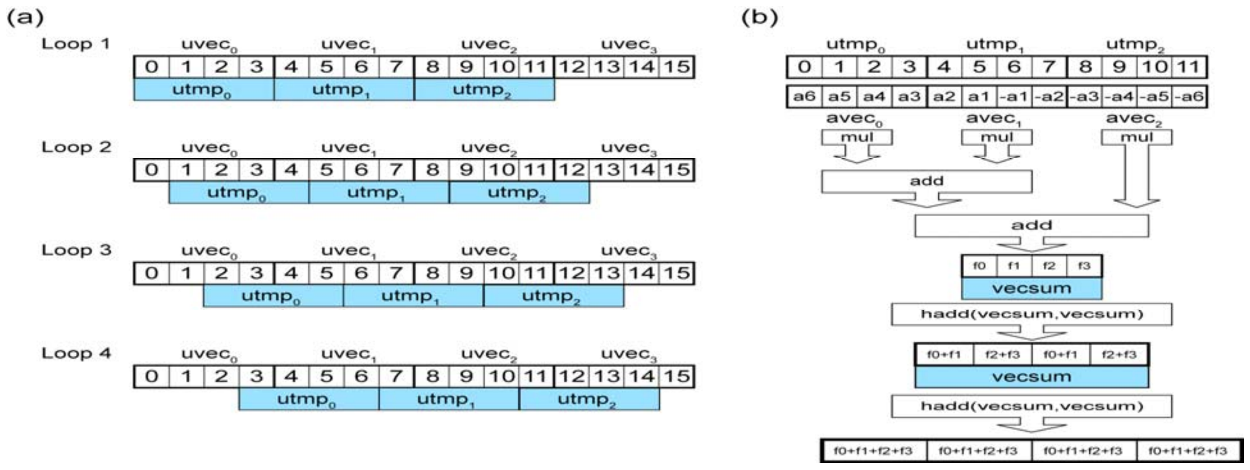

Figure 11. (a) Loop unrolling for seismic code and (b) SIMD calculation using the Intel SSE3 hadd instruction.

```
pattern$_0$,pattern$_1$,pattern$_2$
for(i=istart;i<iend;i+=4){
  Loop one:
  utmp$_k$=uvec$_k$ (k=0,1,2);
  compute(bD[i+5]);
  load uvec$_3$;
  Loop two:
  shuffle(pattern$_0$);
  compute(bD[i+6]);
  Loop three:
  shuffle(pattern$_1$);
  compute(bD[i+7]);
  Loop four:
  shuffle(pattern$_2$);
  compute(bD[i+8]);
  store
 {bD[i+5],bD[i+6],bD[i+7],bD[i+8]}
{uvec$_0$,uvec$_1$,uvec$_2$}=
{uvec$_1$,uvec$_2$,uvec$_3$}  //data reuse
}
```

The speedup of this SIMDization can be analyzed as follows. The original program requires 19 operations to get one bD while the SIMDized program operates 10×4 = 40 to get four bDs. Thus the speedup is 1.9. Memory access can also be analyzed as follows: The original program requires 15 memory accesses to get one bD, while the SIMDized program needs only one load, nine shuffle and one store to get four bDs—in each round, we reuse three vectors from the last round (see the last statement of the above pseudo-code), and consequently only one vector u_4 needs to be loaded. After obtaining four bDs, the SIMD code writes back it as a vector of
$\{bD[i+5], bD[i+6], bD[i+7], bD[i+8]\}$.
The memory access is thus reduced by a factor of over 5. In summary, this solution not only utilizes SIMD and loop unrolling but also enhances data reuse.

3.2. Performance Tests

We have tested the performance of SIMD parallelization for the seismic code on Intel dual quadcore Xeon with 2.33 GHz clock, which has 32 KB L1 data cache per core with shared 2×4 MB L2 cache and 16 GB DDR2 memory. The processor also includes 128-bit wide registers and a separate register for data movement. It also features Streaming SIMD Extension 2 (SSE2) and Streaming SIMD Extension 3 (SSE3) instructions to support SIMDization.

Figure 12(a) shows the running time (to perform one of the major for loops for the stencil computation in the x direction) of the FDTD simulation without and with SIMDization for different numbers of threads for 400^3 grid points (the results are averaged over ten runs), while Fig. 12(b) shows the corresponding speedup due to SIMDization. Our SIMD solution achieves 2-3 fold speedup for all numbers of threads. Figure 12(c) shows the running time of the code segment to perform stencil computation in the x direction (mainly consisting of three for loops) without SIMD optimization, with one of the three loops SIMDized, and with all three loops SIMDized. The figure shows that our SIMD solution reduces the running time when one third of computation is SIMDized, while it cannot improve the performance when SIMD is implemented in all three loops.

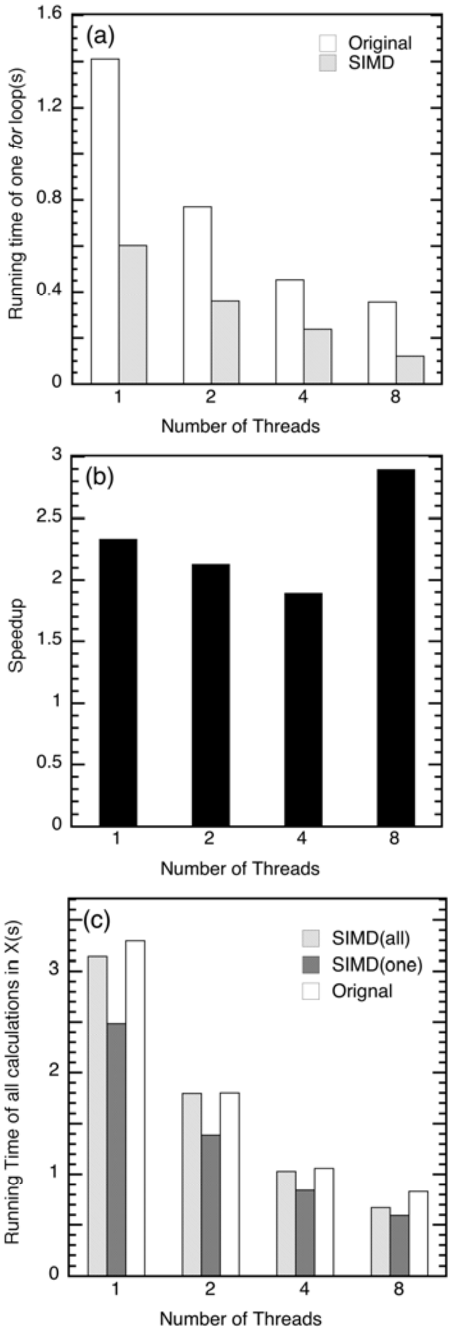

Fig 12. (a) Running time of seismic code with and without SIMD for different numbers of threads. (b) Speedup due to SIMDization. (c) Running time of the code segment for stencil computation in the x direction without SIMD, with SIMD in one calculation and with SIMD for all the three calculations in x direction.

Obviously, SIMD optimization on the seismic code is not as effective as on pLBM, for which several reasons are conceivable. The first is the problem size—the pLBM is tested with the largest size of 64^3 while the seismic code is with 400^3. The large problem size has caused a high TLB (translation lookaside buffer) miss ratio of 0.2% on the Intel platform and thereby decreased the performance. The large problem size also causes large stride memory access when transposing the array in the y and z directions, which further degrades the performance of the whole program. Another reason is the hardware restriction. The Intel platform has only eight vector registers, which cannot feed 30 vectors required for each round of computation in the seismic code. This may cause frequent exchanges of data between vector registers and main memory, which greatly decreases the performance. This also explains the performance decrease when the whole x-direction computations are SIMDized instead of only one-third of it. On the contrary, on Cell BE based PlayStation3, there are 128 vector registers per SPE, which is more than enough to satisfy the needs of pLBM, and accordingly the SIMDized pLBM has achieved a better performance.

4. Conclusion

In summary, we have developed a hierarchical parallelization framework that combines spatial decomposition, multithreading with critical section-free, dual representation, and SIMDization using translocated statement fusion, vector composition and vectorized data layout transformation. For diagonally dominant stencil computations such as LBM, the SIMD parallelization has been found highly effective on multicore platforms such as a Cell BE based PlayStation3 cluster. However, for highly off-diagonal stencil computations such as the high-order FDTD code, the SIMDization has proved less effective on a Intel quadcore platform due to high TLB miss, large stride memory accessing, and lack of vector registers. This points out the need for further optimizations to increase the data locality of the high-order stencil computation.

This work was partially supported by Chevron—CiSoft, NSF, DOE, ARO, and DTRA. Scalability and performance tests were carried out using the Playstation3 cluster at the Collaboratory for Advanced Computing and Simulations of the University of Southern California, the BlueGene/L at the Lawrence Livermore National Laboratory, and BlueGene/P at the Argonne National Laboratory. We thank Dr. Lin H. Yang for his help on the BlueGene/L benchmark and the Argonne Leadership Computing Facility for their help on the BlueGene/P benchmark.

References

[1] L. Renganarayanan *et al.*, in *Proc. of Int'l Parallel and Distributed Processing Symp. (IPDPS)* (IEEE, 2007).

[2] A. Sawdey and M.T.O'Keefe, in *Proc. of Int'l Workshop on Languages and Compilers for Parallel Computing* (LCPC, 1997).

[3] A. Taflove and S.C.Hagness, *Computational Electrodynamics: The Finite-Difference Time-Domain Method, 3rd Ed.* (Artech House Publisher, 2005).

[4] A. Nakano, P. Vashishta, and R. K. Kalia, Comput Phys Commun **83**, 181 (1994).

[5] R. Hararlick and L. Shapiro, *Computer and Robot Vision* (Addision Wesley, 1992).

[6] M. Paleczny, K. Kennedy, and C. Koelbel, Technical Report, Rice University, 94509S (1994).

[7] R. Schreiber and J. Dongarra, Technical Report, University of Tennessee (1990).

[8] F. Desprez *et al.*, Journal of Information Science and Engineering **14**, 167 (1998).

[9] K. Kamil *et al.*, in *Proc. of Workshop on Memory System Performance (MSP)* (ACM, 2006).

[10] A. J. C. Ladd and R. Verberg, J Stat Phys **104**, 1191 (2001).

[11] L. Peng *et al.*, Lecture Notes in Computer Science **5168**, 763 (2008).

[12] A. Stathopoulos *et al.*, Comput Sci Eng **2**, 19 (2000).

[13] F. Shimojo *et al.*, Phys Rev B **77**, 085103 (2008).

[14] J. A. Kahle *et al.*, IBM Journal of Research Development **49**, 589 (2005).

[15] D. A. Bader *et al.*, Parallel Computing **33**, 720 (2007).

[16] S. Williams *et al.*, in *Proc. of Int'l Parallel and Distributed Processing Symp. (IPDPS)* (IEEE, 2008).

[17] G. Rivera and C. Tseng, in *Proc. of Supercomputing (SC00)* (IEEE/ACM, 2000).

[18] M. Frigo and V. Strumpen, in *Proc. of Annual Int'l Conf. on Supercomputing* (ACM, 2005).

[19] D. Wonnacott, in *Proc. of Int'l Parallel and Distributed Processing Symp. (IPDPS)* (IEEE, 2000).

[20] S. Krishnamoorthy *et al.*, in *Proc. of SIGPLAN* (ACM, 2007).

[21] K. Datta *et al.*, in *Proc. of Supercomputing (SC08)* (IEEE/ACM, 2008).

[22] D. Nuzman, I. Rosen, and A. Zaks, in *Proc. of Conf. on Programming Language Design and Implementation (PLDI)* (ACM, 2004).

[23] M. Gschwind *et al.*, IEEE Micro **26**, 10 (2006).